sfiCAN: a Star-based Physical Fault-Injection Infrastructure for CAN networks

David Gessner, Manuel Barranco, Alberto Ballesteros, Julián Proenza, *Senior Member, IEEE*

***Abstract*— The dependability requirements of distributed embedded control systems demand appropriate evaluation techniques. Requirements of embedded systems are often tested by means of fault injection. However, for the Controller Area Network (CAN) the potential of this technique has not been fully exploited. This paper presents *sfiCAN*: a novel physical fault-injection infrastructure that relies on a CAN-compliant hub that allows to inject faults independently into each node's transmitted or received bits, thereby recreating fault scenarios beyond the capabilities of other injectors for CAN. Notably, it is the first injector that is able to test the behavior under inconsistency scenarios of arbitrary software for CAN nodes and the first that makes it possible to inject faults that may lead to integrity errors without requiring any modifications to the nodes' software or CAN controllers. Moreover, sfiCAN allows to remotely and flexibly configure the fault injection and to retrieve accurate information about the subsequent behavior of the nodes.**

I. Introduction

The Controller Area Network (CAN) protocol [1] is a mature, low cost and robust technology that, just like Ethernet, is nowadays used more than ever. It is one of the most widely used field buses in distributed embedded control systems, and still today new standards for CAN are appearing, such as the CAN with flexible data-rate (CAN-FD) and ISO 11898-6. In fact CAN is penetrating old and new markets and new high-volume applications are potentially upcoming [2].

In the automotive domain CAN is the most widely adopted network technology [3], and new CAN-based applications and protocols are expected to be introduced [4]. In fact, although the use of CAN in the most critical automotive applications is still controversial, standards like AUTOSAR establish it as one of its fundamental technologies [5], and there is interest in integrating CAN with newer fieldbuses [6], [7] like FlexRay [8], which are used as high-speed backbones given their increased bandwidth. In part this interest in CAN is due to the current economic situation, which makes companies reluctant to invest in newer—but more expensive—technologies. Additionally, some practical shortcomings have been identified regarding the development of distributed systems with newer fieldbuses such as FlexRay. Finally, regulations in the United States and the European Union ensure the future use of CAN in automotive by making it a recommended or even mandatory protocol for some parts of an automobile [9], [10].

D. Gessner, M. Barranco, A. Ballesteros and J. Proenza are with the Systems, Robotics and Vision group (SRV), Departament de Matemàtiques i Informàtica, Universitat de les Illes Balears (UIB), 07122 Palma de Mallorca, Spain (e-mail: davidges@gmail.com, manuel.barranco@uib.es, a.ballesteros@uib.es julian.proenza@uib.es)

The potential growth of CAN is also seen in other domains. For instance, in aerospace applications CAN is being integrated with other networks [11], as it is reflected in the development of the ARINC 825 standard. Specifically, this standard is devoted to making CAN suitable for flight safety-critical systems in future aircrafts, where this technology is envisaged to be used as either a primary or a secondary network. Moreover, CAN is used as the underlying technology in other safety-related protocols like SafetyBUS p, DeviceNet Safety, and the forthcoming CANopen Safety EN 50325-5. Finally, the academia has been especially concerned with CAN during the last two decades, and it has proposed several solutions to improve its real-time and dependability features, e.g. [4], [5], [12]–[30].

This interest in providing new CAN-based applications and protocols raises the necessity of an adequate infrastructure for testing them. Specifically, a test infrastructure that is able to inject faults and then observe the system's behavior, i.e. a *fault injector*, is widely accepted as a fundamental verification technique to thoroughly evaluate how highly-reliable fault-tolerant and highly-available systems respond to faults.

For many industrial applications, an adequate fault injector would be one that could physically inject faults at the level of the network, i.e. a *physical layer fault injector*, and that at the same time could test production software executing on CAN nodes. This is so because of several reasons. First, testing of real systems or prototypes by means of physical fault injection can provide more accurate and realistic results than other techniques, such as simulation-based ones [31]. In fact, the interest in prototype-based fault injection is especially relevant not only in CAN, e.g. [32], but in other fieldbuses like the Time-Triggered Protocol (TTP), FlexRay and Ethernet, e.g. [33]–[36]. Second, in certain domains maintenance costs can match or even surpass development costs and, thus, it is already necessary to comprehensively test the software of a system prototype prior to production. This is the case of the automotive industry [5], in which CAN is used extensively, and where a significant amount of recalls are due to software errors as recently reported by the United States National Highway Traffic Safety Administration (NHTSA)[1]. Finally, many standards for safety-related systems highly recommend fault injection as a validation technique. Examples are the ISO 26262 [5] for automobiles and the generic standard for safety-related electronic systems IEC 61508 [37], which makes fault injection even mandatory in some cases. Note that many safety-related systems rely on a network who's integrity cannot

[1] A flat file of the NHTSA safety recall database can be obtained at http://www-odi.nhtsa.dot.gov/downloads/

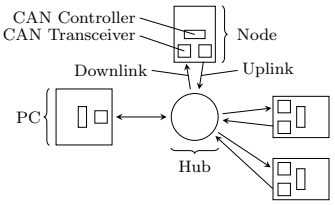

Fig. 1. Fault-injection architecture

be guaranteed and, then, safety is achieved by adding the appropriate mechanisms at other levels of the architecture. Injecting errors at the lowest layers of the communication stack to induce faults at other system levels allows to check the production software that implements those mechanisms.

Despite this evident need, to the authors' best knowledge, the full potential of prototype-based physical fault injection has not yet been fully exploited for CAN, e.g. there is no tool available to test arbitrary production software executing on CAN nodes that allows to inject faults causing inconsistencies [38], [39]. To fill this gap, this paper presents the design and implementation of *sfiCAN*, the first *Star-based physical Fault Injector for CAN*, which provides testing capabilities beyond any other fault-injector previously proposed for this protocol. A preliminary and incomplete version of sfiCAN was introduced in [40].

As depicted in Fig. 1, sfiCAN relies on a star topology whose hub implements a coupling schema based on the one of CANcentrate [41]. This coupling schema is transparent for the nodes, so that from their point of view the hub is logically equivalent to a CAN bus, and the star can be built using *Commercial Off-the-Shelf* (COTS) components. Moreover, this coupling schema makes it possible to monitor and alter, bit by bit, the contribution each node transmits and/or receives, thereby yielding important *testability* [34] benefits.

First, it provides high *controllability* to precisely adjust when and where to inject faults, thereby enabling the injection of complex scenarios such as those leading to data inconsistencies [38] or integrity errors, i.e. errors that lead some nodes to accept spurious frames. This is so because the hub distinguishes every single bit each node transmits/receives and, thus, it can inject faults independently in the individual bits transmitted or received by each node. To inject faults with such high spatial and time resolution is hardly possible with a traditional bus-based injector. The reason for this is that the wired-AND function of the CAN bus irreversibly mixes all nodes' contributions and, thus, it is not possible to use a single device to observe and inject simultaneous errors into the contributions of different nodes. Second, it provides high *observability* as it can log each injected fault and determine, thanks to its central position, the bit stream issued by each node in response to that fault for a later analysis. This information is complemented with a software logger, embedded in each node application, which gathers additional information concerning the node's behavior. Third, the hub makes it easier to carry out *remote testing* [34] from a personal computer (PC)-based management station connected to a dedicated port of the hub. Since the hub allows the station to communicate through it via CAN, the station can configure and coordinate the fault injector and the loggers remotely without needing an alternative network, and it can seamlessly exchange information with the hub during the execution of a fault-injection experiment. Fourth, sfiCAN can be used to transparently carry out an *online testing* [34] or field monitoring of CAN networks that rely on the star-based architectures of CANcentrate and ReCANcentrate [27]. Note that to offer this capacity as an addition to *offline testing* is interesting since highly-dependable systems operate during large periods of time and, like other technologies, e.g. [42], [43], CAN can adopt a star topology as a means to be fit for real-time highly reliable applications [19], [27], [28]. Finally, centralizing the injection of faults reduces the complexity and the cost of the whole fault-injection infrastructure as there is no need to implement a fault injector locally at each node.

The paper is organized as follows. Section II describes previous work on fault injectors for CAN and thoroughly compares them with sfiCAN. Section III explains the basics of sfiCAN, whereas Section IV focuses on how the centralized fault injector is configured and operates. Section V overviews a prototype implementation of sfiCAN and Section VI describes a set of fault injection experiments performed with this prototype that shows the potential of sfiCAN. Finally, Section VII concludes the paper.

II. Related work

Table I compares some testability-related features of sfiCAN and the most relevant and representative fault injectors that can be used for CAN. These features are classified into three blocks labeled as CONTROLLABILITY, OBSERVABILITY and COMPATIBILITY.

To better understand this comparison it is necessary to recall that, thanks to its privileged view of the communication, the hub of sfiCAN can easily distinguish, monitor and change, bit by bit, the stream each node transmits and/or receives. In contrast, the other fault injectors available for CAN rely on a bus topology and, thus, do not have these capacities. This is so because the CAN bus implements a wired-AND function that irreversibly mixes the contributions of all the nodes, so that it is not longer possible to distinguish each contribution within the resultant (global) traffic. Due to this, sfiCAN outperforms the other CAN fault injectors in many aspects.

First, note that sfiCAN injects errors with both a high spatial and a high time resolution, whereas the other injectors present limitations in achieving both kinds of resolution (see column "Inj. Resol."). One of the key points is that in order to inject faults with the same spatial resolution as sfiCAN in a bus, it would be necessary to attach to each node a fault-injection unit able to alter the data that its node transmits/receives locally. This is the case of a few fault injectors such as for example CANoe (CANalyzer) [44] and IFIs [45]. However, these solutions based on distributed hardware fault injectors are difficult to implement and are highly invasive as shown in the column "Invasiv."

In principle, injecting faults with a high time resolution in a bus is not an issue. Even fault injectors that consist of a single device attached to the bus, such as CANstress [46], can inject

errors in different parts of the bit time. Nevertheless, many bus-based approaches can only inject faults with a time granularity equal to the CAN frame, i.e. the user cannot specify in which bit/s to inject errors. This is so because they do not inject errors directly into the channel, but at other levels. This is the case of CANoe, which injects errors in the application by corrupting the nodes' memory; or the case of simulation-based (see column "Type") injectors like Castor/Pollux [47], which inject at the registers and automata of CAN controllers synthesized within Field-Programmable Gate Arrays (FPGAs). In the case of sfiCAN, it would be possible to even inject with a granularity finer than the bit time, since the hub includes a module that keeps it synchronized with the bit stream and which could be used to distinguish between the different time quanta that compose each bit.

Another aspect in which sfiCAN is superior is its capacity for allowing the user to specify high-resolution and complex conditions to trigger the start/end of injections (see column "Trig. Resol. and Complex."). Specifically, the privileged hub position allows it to define triggers based on a holistic vision of the streams each node transmits or receives. In contrast, due to the difficulties to distinguish each node's contribution in a bus, almost all the other fault injectors that inject errors in the channel present a low spatial trigger resolution, and only allow specifying triggers in terms of the resultant (global) traffic observed in the bus, e.g. CANstress. In fact, the only two exceptions to this kind of injectors are NTCAN [48] and IFIs, which partially overcome this limitation by coordinating the set of fault-injection units, one per node, they rely on. In NTCAN each unit is provided with an external trigger input/output that allows to interconnect, and then to trigger, them in cascade. This mechanism is less flexible and cost-effective than to evaluate each node contribution directly from the hub. Moreover, triggers in NTCAN can only be based on the contribution each node transmits, but not on the signal they receive. Regarding the IFIs, it requires the user to know the contribution of each node in advance and, then, to manually trigger the injection of errors off-line based on a pre established traffic that is not always easy to forecast. Moreover, it therefore does not allow to test arbitrary software, but only software whose traffic pattern is deterministic and can be known in advance (see "Prod." column, which specifies the compatibility with production Software, Sw, and Hardware, Hw). On the other hand, approaches that do not inject errors in the channel but at higher levels, e.g. CANoe, do not present trigger spatial limitations, but can only provide a low time trigger resolution with a time granularity equal to the frame. As concerns simulation-based injectors, they only implement triggers that are random or that are specified in terms of a probability distribution, e.g. Pollux and RTaW-Sim [30].

The combination of both features, i.e. the high injection resolution and the high trigger resolution/complexity of sfiCAN, allows it to inject fault scenarios in the channel whose complexity is far beyond the capacities of any other injector. In fact, sfiCAN is able to inject scenarios whose complexity not only induces faults at the Logical Link Control (LLC) layer, but at the application (see "Inj. Lay" column). This allows to test the behavior under complex fault scenarios of unmodified production software. Moreover, it can induce byzantine (arbitrary or malicious) faults at that level, which are the harshest kind of faults that can be experienced, e.g. integrity errors. Note that other injectors like NTCAN, CANstress or RTaW-Sim can also induce faults at the application from the channel, but with a lower severity, e.g. incorrect computation or performance failures. The only bus-based approach that can induce byzantine faults by injecting in the channel is IFIs, but it presents the strong limitations mentioned above. Another alternative to inject byzantine faults is to do it directly at the application level, e.g. like CANoe. This may provide a tighter control on the fault injection at that level, but at the expense of being more invasive. In this sense, note that sfiCAN could be extended with a set of software units placed at the nodes to inject directly at the application if needed.

The trigger features of sfiCAN by their own also allow to flexibly specify start/end conditions to inject permanent, intermittent and transient faults, i.e. sfiCAN provides a tight and easy control of the frequency with which the injected errors cycle between the active and the dormant states. Other approaches can inject errors that exhibit these timing behaviors (see column "Time mod."), but they do not provide the same degree of control and flexibility.

Another advantage derived from sfiCAN's trigger capacities is that it can reproduce and repeat specific fault scenarios (see column "Deter.", i.e. determinism). This can be barely achieved by injectors with lower trigger resolution/complexity, e.g. CANstress, or that would need to coordinate several fault-injection units to implement trigger conditions based on the contribution of different nodes, e.g. CANoe or NTCAN.

Apart from improving the mentioned injection and trigger aspects, the hub's privileged view of sfiCAN also enhances observability. In this sense note that even if column "Impl." indicates that a given injector does not provide any sort of mechanism to log and retrieve data concerning the results of the fault-injection experiment, we still indicate what would be the observability features that injector could potentially achieve. As can be seen in the column that specifies the observability resolution, "O. Resol", the hub can potentially log, for every single bit that is broadcast, the logical value each node transmits/receives. The only fault injector that could potentially achieve such a resolution is the IFIs. For that purpose, it would need to implement a log unit in each node that logs the bits its node transmits/receives. But even in that case, it would be very difficult to match the different nodes contributions in order to determine their simultaneity and, then, to analyze what is the contribution issued by each node in response to a given injected error.

As concerns the layer from which sfiCAN can retrieve information (column "O. Lay") note that its hub can log information related to the LLC layer. However, as already mentioned, sfiCAN implements a software logger, which is attached to each node in a minimal invasive way (see Section V), to track data at the level of the interface between the application and the CAN controllers. A similar logger can be implemented to retrieve data concerning the application itself.

The last advantage of sfiCAN is that its hub includes a dedicated port to connect a personal computer (PC)-based

management station. This paves the way to carry out remote testing (see column "Remote"). On the one hand, the hub itself allows the station to communicate, via CAN, with the fault injector and the loggers; this enables the station to remotely configure, coordinate and retrieve information from them without needing an additional network. On the other hand, the hub can create two separated communication domains, so that the station can exchange information with the hub without interfering with the communication among the nodes during a given fault-injection experiment. Some bus-based approaches also provide remote testing, but either do include an additional network to communicate the station with the injectors and loggers, e.g. IFIs, or they do not allow a seamless integration of the management and the experiment traffics, e.g. CANoe.

Finally, note that the main disadvantage of sfiCAN is that for the hub to be able to distinguish the nodes' contributions, each node connects to the hub by means of a separated uplink and downlink. As explained later, this connection requires to include an extra transceiver per node like in CANcentrate [41]. This limits the applicability of sfiCAN to arbitrary hardware configurations of systems in production. However, this is a relative disadvantage when compared with the other injectors, as all of them (possibly excluding CANstress) present some sort of incompatibility (column "Prod."). Moreover, conversely to other injectors, sfiCAN is totally compatible with COTS hardware components.

The reader is also referred to other bus-based fault injectors less powerful than those of Table I. Examples of physical injectors are [49], [50], [51] and [52]. The first one is a middleware-based distributed fault injector, which is devoted to inject transient faults that provoke frame transmission/reception and control feedback delays, as well as detectable corrupted messages. The other three injectors resemble CANstress, but either induce less harmful failures at the application (or can induce no fault at this level), present lower injection resolution and trigger complexity, can only inject intermittent faults, or cannot force specific error scenarios. Concerning simulators, some authors [53] have used a generic, i.e. not specific to CAN, fault injection tool for hardware description languages (HDLs), called SINJECT [54]. It provides features similar to those offered by Pollux, but it can hardly induce faults at the application level. Another injector is proposed in [55] which uses a set of MATLAB/Simulink models of both the CAN network, where the faults are injected, and the behavior of a vehicle relying on that network. Unfortunately, the network model is an abstraction above the frame level and can thus not be used to study the consequences of bit-level errors.

III. Design of sfiCAN

SfiCAN is composed of a set of parts which cooperatively work to carry out a *fault-injection experiment*, i.e. an experiment during which the behaviour of a given target system is analysed when it is forced to deal with errors provoked by faults.

The central element of sfiCAN is a hub to which the nodes of the system and a PC-based management station are connected. This hub implements a coupling schema based on the one of CANcentrate [41]. On the one hand, the hub provides a CAN network that allows to distribute the different components of sfiCAN as a set of *Network Configurable Components* (NCCs), i.e. components whose operation can be configured and coordinated remotely through the network. The NCCs are located within the hub, as well as on the system's nodes. On the other hand, the hub's coupling schema allows to implement advanced fault-injection and logging features within the hub itself.

A text file called *fault-injection specification*, which is stored in the PC, contains the description of the faults to be injected in an experiment. The PC configures the NCCs in accordance with this specification, triggers the execution of the experiment and, once it is finished, retrieves the information logged by the NCCs.

Next all these concepts are explained in more detail.

A. Types of physical faults in CAN

A CAN network includes different hardware components ranging from cables to CAN controllers. Although components may suffer from a high variety of faults, e.g. shorted cables, damaged connectors, etc., most of these faults manifest as errors that corrupt the logical value of the bits being transmitted or received by each node. Thus, sfiCAN does not inject faults physically at components, but instead injects the different types of erroneous bits these faults would generate in the channel. Specifically, given that in CAN the two possible bit values are called dominant and recessive (a dominant bit '0' prevails over a recessive bit '1' [1]), these errors are called stuck-at-recessive, stuck-at-dominant and bit-flipping [41]. A stuck-at bit stream consists of a sequence of consecutive bits of the same logical value, whereas a bit-flipping stream is a sequence of bits that randomly alternates from recessive to dominant and vice versa.

B. sfiCAN basics

Fig. 1 shows how each node is connected to the hub of sfiCAN by means of a dedicated link comprised of a separated *uplink* and *downlink*. The node's CAN controller connects to two COTS CAN transceivers as in CANcentrate [41], in which one transceiver is used to transmit through the uplink and another one to receive from the downlink.

Fig. 2, which depicts the internal structure of the hub, shows how each node contribution, B_i, is received through the corresponding uplink. When no fault is being injected, each contribution propagates from its *uplink multiplexor*, $umux_i$, to the *Coupler Module*, which couples all of them by means of an AND gate. Then, the *resultant coupled signal*, B_0, passes through each *downlink multiplexor*, $dmux_i$, and is broadcast back to the nodes via the downlinks. Since this coupling is done in a fraction of the bit time, the frame observed at B_0, i.e. the *resultant frame*, is the same as in a CAN bus, thus making the hub transparent for the nodes. However, in contrast to a bus, a star topology allows the hub to distinguish the signal each node locally transmits and receives. Thus, as opposed to bus-based fault injectors, sfiCAN can inject channel faults

TABLE I

COMPARISON OF FAULT INJECTORS

	CONTROLLABILITY					OBSERVABILITY			COMPATIBILITY			
	Inj. Resol.	Trig. Resol. and Complex.	Inj. Lay	Deter.	Time mod.	Impl.	O.Resol	O.Lay	Type	Invasiv.	Prod.	Remote
sfiCAN	Node tx/rx Bit	Based on **holistic vision of each node's tx/rx bit**. **Up to N independent** conditions in terms of: frame's type, field and bit; bit-pattern; number of pattern occurrences and offset	App (byzantine) LLC	Yes	Permanent Intermittent Transient	Yes	Bit Node tx/rx	App LLC	Phy	Extra txrx per node	Sw (COTS Hw)	Yes
NTCAN	Glob. traffic Bit	Based on individual nodes' tx bit. Up to 5 independent conditions in terms of: frame's field and bit; bit pattern; offset; and external input for injecting in cascade	App (inc. comp) LLC	Barely	Permanent Transient	No	Frame Glob. traffic	LLC	Phy	Inj. unit synthesized in each node	Sw	No
CANoe	Node tx/rx Frame	Based on individual nodes' tx/rx frame: its ID and payload	App (byzantine) LLC	Barely	Permanent Intermittent Transient	Yes	Frame Node tx/rx	App	Phy Sim	Inj. unit attached to each node. Sw modif.	Hw	Partial
CAN stress	Glob. traffic Bit fraction	Based on broadcast frame: type, start/end and bit	App (inc. comp) LLC Phy	Barely	Permanent Intermittent Transient	Yes	Bit fraction Glob. traffic	LLC Phy	Phy Sim	—	Sw Hw	No
IFIs	Node tx/rx Bit	Based on holistic vision of each node's tx/rx bit, but specified offline for a traffic known in advance.	App (byzantine) LLC	Yes	Permanent Intermittent Transient	No	Bit Node tx/rx	LLC	Phy	Inj. unit attached to each node	No	Partial
Castor Pollux	Glob. traffic Frame	Random	App (inc. comp) LLC	No	Permanent Intermittent Transient	No	Frame Glob. traffic	LLC	Sim	Inj. units synthesized in each node	No	—
RTaW-Sim	Glob. traffic Bit	Probability distribution	App (perform.) LLC	No	Transient	Yes	Frame Glob. traffic	App	Sim	—	No	—

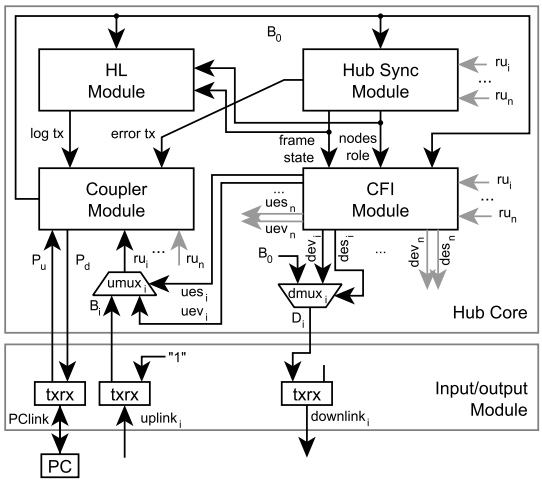

Fig. 2. Basic structure of the hub

that affect some nodes but not others and can determine the response of each individual node to the injected faults.

The hub also has a dedicated port to which the PC-based management station is connected using a COTS CAN controller board. The corresponding link is not separated into an up- and downlink since no faults are injected into this link. This extra connection allows the PC to remotely configure and coordinate the different Network Configurable Components (NCCs). The NCCs are the elements that actually inject faults and log information concerning these faults and their consequences on the system. The different types of NCCs are the *Centralized Fault Injector* (CFI), the *Hub Logger* (HL) and the *Node Loggers* (NLs).

The CFI and the HL are hardware modules located within the hub (see Fig. 2). The former is the responsible for injecting faults at the signal transmitted or received by each node; whereas the HL logs information about each frame broadcast during the fault injection experiment. For each frame, the HL logs its source port, identifier, data field and, if the frame is affected by an error, also the location of that error (frame field and bit number). Regarding each NL, it is a piece of software attached to each node's application that logs information about the internal state of the node during an experiment. The NL gathers the frames the application successfully transmits or receives, and the value of the *Transmission/Reception Error Counters* (TEC/REC) [1] of the CAN controller. This information makes it possible to determine which frames the node rejected and the state of the controller itself, i.e. *error-active*, *error-passive* or *bus-off* [1].

The responsible for configuring and coordinating the NCCs is a software called *Fault Injection Management Station* (FIMS), which executes on the PC. The FIMS contains the fault-injection specification file and uses a so-called *NCC protocol* on top of CAN (Section III-D) to configure each NCC accordingly. It also uses this protocol to compel the NCCs to start the experiment at the same time and, once the experiment is finished, to collect the data monitored by each NL and the HL.

C. Internal structure of the hub

The hub is composed of the *Input/Output Module* and the *Hub Core Module* (Fig. 2). The former includes a set of COTS transceivers that translate the physical signals received from the nodes and the PC into a logical form and vice versa. The Hub Core is the part that actually implements the coupling, fault-injection and logging mechanisms. It includes the *Coupler Module*, which was explained before, the *Hub Sync Module*, and the modules that implement the *Centralized Fault Injector* (CFI) and the *Hub Logger* (HL).

To understand the purpose of the *Hub Sync Module*, note that CAN nodes have a quasi-simultaneous view of every bit on the channel and that they are synchronized at bit and frame level, i.e. for each bit being broadcast they agree on its logical value, its location within the frame field, etc. This agreement is the basis of important CAN mechanisms such as the *bit-wise arbitration*, which determines the *role* being played by each node (transmitter or receiver), and the *error signaling*, which allows any node to globalise any local error it detects. This error signaling mechanism keeps nodes synchronized even in the presence of errors and is supposed to provide *data consistency* [1], i.e. to ensure that every frame is either accepted or rejected by all nodes.

In order to inject faults at specific bits and to log information about the contribution of each node to each frame, the hub needs to stay synchronized with the nodes and know their roles. This is accomplished by the Hub Sync Module, which synchronizes at the bit and at the frame level with the *resultant frame* being broadcast at B_0, and which deduces the role of each $node_i$ by analyzing its contribution at the corresponding ru_i signal. To keep synchronized, the Hub Sync Module both signals and globalises (as any CAN node would do), by means of the *error tx* contribution, any error it detects on the resultant frame. As a result, the Hub Sync Module provides the CFI and the HL modules with a set of signals, *frame state*, that specify the meaning of the bit being broadcast at B_0, i.e. the current state of the *resultant frame*, and a set of signals called *nodes role* that codify each node's role.

The CFI Module stores the configuration of the fault-injection experiment and injects the corresponding errors at the appropriate bits of the specified uplinks or downlinks. A given error consists in a single bit whose value (recessive or dominant) deviates from what is expected according to the current state of the resultant frame, e.g., a recessive stuff bit when a dominant stuff bit is expected. Each erroneous bit can be injected independently in each uplink or downlink by means of a dedicated *uplink multiplexor* (*umux*) or *downlink multiplexor* (*dmux*), respectively. A given $umux_i$ receives as an input from the CFI Module an *uplink error selection* signal, ues_i. This signal decides what the output ru_i of the $umux_i$ multiplexer should be, thereby deciding the contribution of the $node_i$ to the Coupler Module, i.e. either the bit transmitted by $node_i$ (B_i), or the erroneous bit to be injected (the *uplink error value*, uev_i). Analogously, the CFI Module can use the signal *downlink error selection* (des_i) of $dmux_i$ to send to the $node_i$ (via D_i) the coupled signal B_0 or the error to be injected in its downlink, i.e. the *downlink error value* (dev_i).

Finally, when the CFI executes a given fault-injection experiment, the HL logs information about the errors signaled by the hub and the frames that are exchanged, correctly or incorrectly, at each uplink port. When the experiment finishes, the HL Module uses its own contribution to the Coupler Module, *log tx*, to send the information it logged to the *Fault Injection Management Station* (FIMS).

D. Operational modes of Network Configurable Components

As said in Section III-B, the FIMS configures and coordinates the NCCs using a so-called NCC protocol, which relies on the exchange of CAN frames through the channel. The CAN identifier field is used to send each frame to either one or all NCCs, using appropriate unicast and broadcast identifiers (NCC IDs) respectively. The data field contains a given command, which can be of three different types: *configuration*, *logging* or *mode change*. The first type is used to tell an NCC what to do during a fault-injection experiment, e.g. what errors the CFI must inject. The second one is used by the FIMS to retrieve log information from the NCCs once the experiment finishes. Finally, a mode change command allows forcing an NCC to switch between different operational modes

Each NCC can work in four different operational modes, namely *idle*, *instruction*, *wait-for-whistle* and *execution*. An NCC in the idle mode does nothing and ignores all commands, except a mode change the FIMS broadcasts to force all NCCs to switch to the instruction mode. The frame containing this command is referred to as the *instruction-mode frame* and the CAN identifier with the highest priority is reserved for it.

During the instruction mode the FIMS can either configure any NCC or retrieve logging information from that NCC, by respectively sending configuration or logging commands within frames with the appropriate NCC ID. The FIMS can also force each NCC to switch to either the idle or the wait-for-whistle mode using the corresponding mode change command.

An NCC in the wait-for-whistle mode behaves as in the idle mode, but apart from changing its mode when receiving an instruction-mode frame, it also reacts to a frame called *starting-whistle frame*. When the FIMS broadcasts this latter frame, all the NCCs in the wait-for-whistle mode switch to the execution mode at the same time. In the execution mode, the NCC carries out the operations it is responsible for during the fault-injection experiment. Note that distinguishing between the idle and the wait-for-whistle modes allows some NCCs to be disabled for a given test.

During the execution mode, the communication between the FIMS and the NCCs is restricted to the broadcast of the *instruction-mode frame*. Since this frame has the highest priority, the FIMS uses it to end the experiment at any time and force all NCCs to enter the instruction mode. This means that the application under test can use all the set of CAN identifiers, except that one. Finally, note that it is mandatory to prevent the CAN controller of every node from reaching the bus-off state during the execution mode. Otherwise the NCC that implements the logger of the node will not receive the instruction-mode frame to finish the experiment.

```
specification = { '[', string, ']', fi_config }
fi_config = value, target, mode, aim, fire,
        cease, [withdraw], [target_frame] ;
value = 'value_type', '=', value_type_value,
        ['value_pattern', '=', {'0'|'1'} ] ;
target = 'target_link', '=', link - 'coupled' ;
mode = 'single-shot' | 'continuous' | 'iterative' ;
aim = 'aim_count', '=', natural,
    'aim_filter', '=', filter_value,
    'aim_field', '=', field_value,
    'aim_link', '=', link,
   ['aim_role', '=', role_value ] ;
fire = 'fire_field', '=', field_value,
       'fire_bit', '=', natural,
       'fire_offset', '=', natural ;
cease = 'cease_bc', '=', natural
    | 'cease_field', '=', field_value,
      'cease_bit', '=', natural ;
target_frame = 'target_frame_filter', '=', filter_value,
    'target_frame_field', '=', field_value,
    'target_frame_link', '=', link,
   ['target_frame_role', '=', role_value ] ;
withdraw = 'withdraw_count', '=', natural,
    'withdraw_filter', '=', filter_value,
    'withdraw_field', '=', field_value,
    'withdraw_link', '=', link,
   ['withdraw_role', '=', role_value ] ;
value_type_value = 'dominant' | 'recessive' | 'pattern' |
     'inverse' ;
filter_value = ('0'|'1'|'x'), {'0'|'1'|'x'} ;
link = 'port0up' | 'port0dw' | 'port1up'
    | 'port1dw' | 'port2up' | 'port2dw'
    | 'port3up' | 'port3dw' | 'coupled' ;
field_value = 'idle' | 'id' | 'rtr' | 'res'
    | 'dlc' | 'data' | 'crc' | 'crcdelim'
    | 'ack' | 'ackdelim' | 'eof'
    | 'interfield' | 'errflag' | 'errdelim' ;
role_value = 'dont_care' | 'tr' | 're' ;
```

Listing 1. Fault-injection specification BNF.

IV. The Centralized Fault Injector

A given fault-injection experiment is described by means of a *fault-injection specification*, which is transferred from the FIMS in the PC to the CFI located at the hub using the NCC protocol just described. List. 1 shows a simplified version of the Backus-Naur Form (BNF) syntax of this specification in the ISO/IEC 14977 standard [56]. Due to space limitations we do not make the semantic restrictions of this specification explicit, but most can be inferred from how the CAN protocol works. A given fault-injection specification is composed of a series of labeled *fault-injection configurations*, each of which includes a set of key-value pairs called *configuration parameters*. When the CFI enters the execution mode, the experiment is carried out by a set of CFI submodules called *fault-injection executors*. These submodules work in parallel and each one of them performs the injection indicated in a different fault-injection configuration.

The configuration parameters indicate what, where, and when to inject. What to inject is given by a *fault-injection value*, e.g. a dominant bit (referred to as `dominant` in List. 1) or a given sequence of bits (`pattern`, with the sequence given by `value_pattern`). Where to inject is designated by a *target link*, e.g. the downlink of port 1 (`target_link = port1dw`).

The specification of when to inject leads the corresponding executor to behave as depicted in the automaton of Fig. 3, in which the states during which errors are injected are highlighted with thick circles. Once the CFI enters the execution mode, each executor is in the *ready* state, meaning that it may start the fault injection. Each executor remains in this state

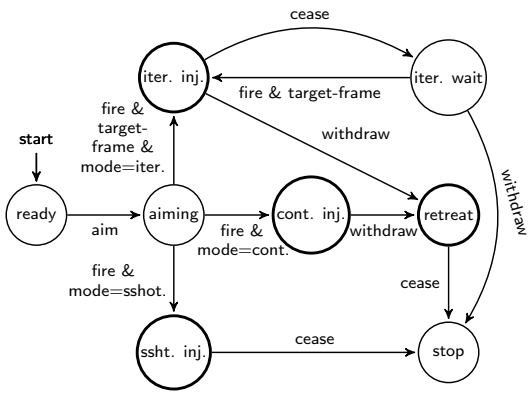

Fig. 3. Fault injector state machine.

until it detects an *aim* condition. This is a condition defined on the traffic observed at a given uplink/downlink, or the coupled signal, that must be satisfied before a fault is injected. This allows starting the injection of errors with respect to a point of reference located in a so-called *start frame*, which is the frame that contained the bits that satisfied the *aim* condition, e.g., the aim condition may be the third reception of a CRC (`aim_field = crc`, `aim_count = 3`) beginning with the prefix '0101' or '0111' (`aim_filter = 01x1`) and detected on the downlink of port 2 (`aim_link = port2dw`) when the node connected to that port is the transmitter (`aim_role = tr`).

An executor detecting this condition progresses to the *aiming* state. From there it proceeds to inject errors following three possible *fault-injection modes*, namely *continuous*, *iterative* and *single shot*. The first one (`mode = cont.`) is used to continuously inject errors to affect several consecutive frames. When an executor in this mode detects the *fire* condition, it switches to the *cont. inj* state and starts injecting errors. This condition is specified as a bit within a given frame field plus an offset that must be observed in the coupled signal In *cont. inj* the executor waits for a condition analogous to the aim, called *withdraw*, which must be satisfied before stopping the injection of errors. When so, the executor continues injecting errors but switches to the *retreat* state, in which it waits until it observes a *cease* condition that indicates that it must stop injecting errors. In the continuous mode, a cease condition consists in a given bit of a frame field that must be observed in the coupled signal. Note that an undefined withdraw is allowed, but results in the injection of a permanent fault.

The iterative mode (`mode = iter.`) is devoted to inject errors only in specific frames and to do so iteratively. The frames where the errors must be injected are specified by means of the *target_frame* condition, whose syntax is analogous to the one of the aim and the withdraw. When the executor detects both the target_frame and the fire condition, it enters the *iter. inj* state during which it injects errors. If the executor detects the cease condition while being in this state, it stops injecting and switches to *iter. wait*. Then it can cycle between the *iter. inj* and the *iter. wait* states depending on which of the mentioned conditions it observes. The executor only stops injecting definitively when it has detected both the cease and the withdraw conditions (if defined), but not exclusively in this order. The cease can be defined in this mode as a bit of a frame field, and also, as a bit count.

The single shot mode (`sshot.`) aims at injecting a sequence of errors within a single frame or in the *Intermission Frame Space* (IFS) [1]. The executor behaves as in the continuous mode, except that it stops injecting when detects the cease condition without the need of observing a previous withdraw circumstance. The cease is defined as in the iterative mode.

Module	Slices	Flip flops	LUTs	IOBs
CFI	5557	3800	7359	87
HL	2058	2089	1935	40
Hub sync	160	77	281	58
Total	7680	15360	15360	173

TABLE II
FPGA OCUPATION SUMMARY

V. IMPLEMENTATION OF SFICAN

We built a prototype of sfiCAN that includes one hub and three nodes. The hub core is implemented using the VHSIC Hardware Description Language (VHDL) and synthesized in a Xilinx Spartan-3 XC3S1000 FPGA, whereas its Input/Output module is built using COTS transceivers. Table II shows that the CFI and the HL occupy the major part of the FPGA resources, as they are the most complex modules and need to store data. The CFI has resources for carrying out an experiment constituted by up to 5 fault-injection configurations. The HL can store log information for 50 frames and implements a transmitter module to report this information to the FIMS.

Each node is composed of COTS components only, i.e. one dsPIC30F6014A microcontroller [57] and two transceivers, one to connect the micro's CAN controller to the uplink and another one to connect it to the downlink. Each link uses one UTP Ethernet cable and a pair of RJ45 connectors, with each uplink/downlink using a different two-wire differential line.

The prototype was implemented orderly by including, debugging, and testing features step by step, which made the validation thorough and easier. For checking the correct operation of the hub, an oscilloscope was used to monitor the bit stream at its different ports and some unused FPGA pins that output the state of its internal modules. The hub's LEDs were used to code the operational mode of the CFI. For each node, the LEDs showed the operational mode of its NL and the value of the data field of each transmitted/received frame. The log provided by the HL and the NLs was also used for debugging.

Finally, the NL is implemented as a layer standing between the application and the CAN controllers. This implementation is minimally invasive from the point of view of the application and makes it possible to gather the logging data mentioned in Section III-B in an almost transparent fashion.

VI. FAULT INJECTION EXPERIMENTS

As already detailed in Section II and Table I, sfiCAN presents testability-related features that allow it to outperform any other CAN fault injector in several aspects. In particular,

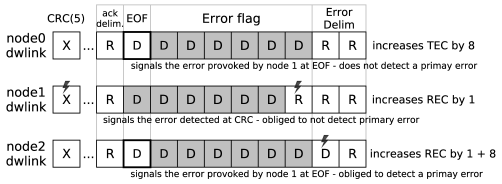

Fig. 4. Unfair Primary Error (UPE) scenario.

the high spatial/time resolution with which sfiCAN injects errors, together with its capacity of implementing high resolution and complex triggers, allows sfiCAN to inject fault scenarios whose complexity cannot be achieved by any of the other injectors. The current section shows a set of experiments that demonstrate this superiority of sfiCAN and discusses some testability benefits derived from it.

The experiments herein presented include between three and six nodes. One of the nodes acts as the transmitter, whereas the other ones act as receivers. For each experiment, the FIMS configures the CFI with the corresponding set of fault-injection configurations and forces all NCCs to switch to the wait-for-whistle mode. Then, the FIMS sends a starting-whistle frame to begin the experiment, and once the experiment has finished, it retrieves the data logged at the hub and the nodes.

In order to give physical evidences for the experiments, we also include some screenshots taken with a Yokogawa DL7440 digital oscilloscope. In general, these captures show how the fine-grained spatial/time resolution of sfiCAN allows to inject complex scenarios involving errors, i.e. they show how sfiCAN alters specific individual bits of the signals transmitted/received by specific nodes in order to force a given scenario. These captures are complemented with a set of tables that summarize the data logged during the experiments.

A. Unfair Primary Error (UPE)

This experiment shows how sfiCAN can inject a complex and realistic scenario that cannot be reproduced by any other fault injector. As it will be detailed later, this is mainly due to its superior injection and trigger resolution.

The experiment uses the potential of the iterative fault-injection mode to create a complex fault scenario that goes beyond the fault-confinement mechanism of CAN called *Primary Error* (PE) [1]. In CAN, a node signals and globalises any error it detects by sending a sequence of 6 consecutive dominant bits called *error flag*, which forces all the other nodes to detect and signal an error too. After its own flag, every node continuously transmits recessive bits, which eventually results in all of them cooperatively transmitting a sequence of recessives called *error delimiter*. If an erroneous bit is detected by all nodes, they all send the flag and the delimiter at the same time, and each node increases its TEC by 8 if it acts as the transmitter, or its REC by 1 if it is a receiver. But if an error is locally detected and then signaled by the flags of some nodes only, the other nodes will respond by sending error flags that hence will be delayed with respect to the first ones. Any node detecting an error locally becomes aware of this situation when it monitors after its error flag a

```
1  [fault injection 1]
2  value_type           = inverse
3  target_link          = port1dw
4  mode                 = iterative
5  aim_filter           = 0
6  aim_field            = idle
7  aim_link             = coupled
8  aim_count            = 2
9  target_frame_filter  = xxxx.xxx1
10 target_frame_field   = data
11 target_frame_link    = coupled
12 fire_field           = crc
13 fire_bit             = 4
14 fire_offset          = 0
15 cease_bc             = 1
16
17 [fault injection 2]
18 value_type          = recessive
19 target_link         = port1dw
20 mode                = iterative
21 ...
22 fire_field          = ackDelim
23 fire_bit            = 0
24 fire_offset         = 7
25 cease_bc            = 1
26
27 [fault injection 3]
28 ...
```

Listing 2. UPE fault-injection spec.

dominant bit, instead of a recessive. Then, it is said that the node detects a Primary Error (PE) and, thus, it further increases its TEC or its REC by 8. This extra penalization causes the TEC/REC of any node suffering from local faults to quickly reach a certain threshold that, then, leads it to switch from the *error-active* to the *error-passive* state, in which its capacity for globalising errors is reduced to minimize its negative impact on the communication.

In this experiment, node0 sends an ordered sequence of one-byte natural values, each in a subsequent frame. Errors are injected in the downlink of the receiving nodes during some of these frames in order to reduce the effectiveness of the PE, as depicted in Fig. 4. First, an error is injected locally in the downlink of node1 during the CRC. This node starts signaling it at the first bit of the *End Of Frame* (EOF) field, causing the other nodes to detect a format error and to signal it at the subsequent bit. After sending its error flag, node1 should have to monitor a dominant bit belonging to the delayed error flags sent by nodes 0 and 2, and thus detect a PE. But a recessive value is injected in its downlink for this bit, so that it does not detect the PE and increases its REC by 1 only. In contrast, although node2 should not have to detect a PE, it is forced to do so by injecting a dominant bit just after it sends its own error flag. Thus, it additionally increases its REC by 8.

This behavior is repeated using the iterative mode to force node2, exclusively, to reach the error-passive state. This is unfair as node1 is affected by two local errors, whereas node2 is only affected by one. Note that errors are injected in alternate frames to not block the bus. For this purpose errors are injected only in frames carrying an odd natural, and the transmitter does not retransmit any frame encountering errors, but tries to transmit a frame with the next natural value.

List. 2 shows part of the 3 fault-injection configurations of this experiment. The first one inverts the 4th bit of the CRC received by node1 in those frames (target frames) car-

	Hub	Node0	Node1	Node2
1	Ok 030#00	Tx Suc 030#00	Rx Suc 030#00	Rx Suc 030#00
2	Er 030#01 (eof(0))	—	—	—
3	error frame	TEC:008 ; tx error → +8	REC:001 ; rx error → +1	REC:009 ; rx error + PE → +1 +8
4	—	Tx Uns 030#01	—	—
5	—	TEC:007 ; tx ok → -1	REC:000 ; rx ok → -1	REC:008 ; rx ok → -1
6	Ok 030#02	Tx Suc 030#02	Rx Suc 030#02	Rx Suc 030#02
7	...	...	...	...
8	Er 030#05 (eof(0))	—	—	—
9	error frame	TEC:127 ; tx error → +8	REC:001 ; rx error → +1	REC:128 ; rx error + PE → +1 +8
10	—	Tx Uns 030#05	—	ERROR PASSIVE
11	—	TEC:126 ; tx ok → -1	REC:000 ; rx ok → -1	REC:127 ; rx ok → -1
12	—	—	—	ERROR ACTIVE
13	Ok 030#06	Tx Suc 030#06	Rx Suc030#04	Rx Suc 030#04

TABLE III
UNFAIR PRIMARY ERROR (UPE) LOG

rying an odd natural in the 1st byte (`target_frame_filter = xxxx.xxx1`) of the data field. Configuration 2 forces node1 to not detect the PE by masking, 7 bits after the ACK delimiter, the dominant it should have received after the 6 bits of its error flag. Although omitted, configurations 1 and 2 define the same target frame conditions. Configuration 3 refers to node2 and is defined similar to configuration 2.

Table III chronologically summarizes the events logged in the experiment. The events related to a given frame are grouped into subtables. Each string with format *iii#dd* represents the frame to which a given event is related, indicating its identifier *iii* and the value *dd* it carries in its data field. The column dedicated to the hub shows when a given frame was successfully broadcast (*Ok*) or not (*Er*). In the latter case, it also indicates the bit and the frame field where the error was encountered (e.g. *eof(0)* refers to the first bit of the EOF) and then, in the next row, it indicates the broadcast of the associated error frame. Similarly, the column of a given node shows the frames it successfully transmits (*Tx Suc*) or receives (*Rx Suc*), its unsuccessful frame transmissions (*Tx Uns*), any change on its TEC/REC, and when it reaches the error-active and the error-passive states. Although the cause of a TEC/REC change is not logged, it can be easily inferred and it is indicated as a comment (*tx/rx error*, *PE*, *tx/rx ok*).

Rows 2, 3 and 4 report what happens when the errors are injected in the first frame carrying an odd natural (01). They confirm that the hub detects an error in the 0th bit of the EOF and that the TEC/REC are increased as expected. Rows 5 and 6 show that the nodes correctly decrease their TEC/REC by 1 when successfully transmitting/receiving a frame containing an even natural. Each node first notifies about a change on the TEC/REC and, then, about the incorrect/correct transmission/reception of the corresponding frame. Rows 8, 9 and 10 confirm that node2 reaches the error-passive state when its REC exceeds the threshold established by CAN [1], i.e. 127; whereas rows 11, 12 and 13 show how this node returns to the error-active state when it successfully receives a frame.

```
1  [fault injection 1]
2  value_type  = inverse
3  target_link = port1dw
4  mode        = single-shot
5  aim_filter  = 0
6  aim_field   = idle
7  aim_link    = coupled
8  aim_count   = 2
9  fire_field  = data
10 fire_bit    = 0
11 fire_offset = 0
12 cease_bc    = 4
13
14 [fault injection 2]
15 value_type    = pattern
16 value_pattern = 111.0011.1000.0101
17 target_link   = port1dw
18 mode          = single-shot
19 aim_filter    = 0
20 aim_field     = idle
21 aim_link      = coupled
22 aim_count     = 2
23 fire_field    = crc
24 fire_bit      = 0
25 fire_offset   = 0
26 cease_bc      = 15
```

Listing 3. Integrity scenario spec.

As pointed out before, no other fault injector can force this scenario. This is so because for this scenario an injector must provide all of the following testability features, which is not the case for the others, as explained in Section II: (1) capacity to inject errors in the reception contribution locally received by different nodes, i.e. node1 and node2 in this experiment; (2) an injection time resolution equal to the bit time, in order to only alter the value of individual bits; (3) high time/spatial trigger resolution for restricting the injection of errors to specific bits, i.e. here the 4th bit of the CRC and the 7th and 8th bits of the EOF of a specific frame; (4) triggers based on a holistic view of the contribution each node transmits/receives, i.e. in this scenario errors locally injected in node2 must affect the same frames in which errors are locally injected in node1;

(5) capacity for repeating, deterministically, the same injection several times; (6) and tight control of the frequency with which errors are injected, i.e. in alternate frames in this example.

B. Integrity error

The purpose of this experiment is not only to inject a scenario whose complexity cannot be reached by any other fault injector, but to show the exclusive capacity of sfiCAN to induce a byzantine fault at an arbitrary application by injecting errors in the channel.

Specifically, this experiment demonstrates sfiCAN's capability of inducing integrity errors (see Section I) at the application level. Note that integrity errors in a real application may occur due to faults in the channel or, more likely, in a CAN controller, e.g. due to the change of a bit in a transmission or reception buffer. Thus, being able to inject faults leading to integrity errors is very valuable to test the behavior of CAN applications and CAN-based protocols when these faults do occur, especially of fault-tolerant applications and systems. Just as an example, note that injecting these faults in systems with high safety integrity levels allows to check whether their additional error detection mechanisms (e.g., the additional CRC of SafetyBUS p) are able to prevent integrity errors from occurring.

In this experiment node0 transmits three times a frame with the hexadecimal value AA in the data field. Node2 receives all frames correctly, whereas node1 is forced to receive an altered version of the 2nd frame. For this, the data and the CRC fields of that frame are changed on the downlink to node1 in such a way that the CRC matches the altered data field. The frame fields are altered by means of two fault-injection configurations, which are shown in List. 3. The first one inverts the value of the first 4 bits of the data field, yielding a data value of $5A$. The second overwrites the value of the CRC with a pre calculated value that corresponds to the modified data.

Figure 5 shows a screen capture of the second frame transmitted by node0. The first row corresponds to the uplink of node0, the second row to the downlink of node1, and the third row to the downlink of node2. Although all three nodes consider the frame to be exchanged correctly (none of them signals an error), the frame accepted by node1 is different from the one transmitted by node0 and accepted by node2. This fact is corroborated by the data received from the NLs, which is shown in Table IV. Each row corresponds to a frame. Each entry indicates whether a given node transmitted or received that frame; and what the identifier and data content transmitted/received was. As the table shows, node1 received and accepted data that was never transmitted by a node.

Note that approaches other than sfiCAN can only provoke an integrity error in CAN by means of software implemented fault injection (SWIFI). This is so because those approaches cannot inject errors in the channel that lead nodes running an arbitrary application to inconsistently receive frames. For that purpose, those injectors would need to alter, on-the-fly, the value of specific bits in the contribution received by a specific subset of nodes. More specifically, as shown in this experiment, no other injector can do this since it cannot: (1) inject errors in the reception contribution locally received by a given node/s; (2) inject with a resolution equal to the bit time; and (3) provide triggers with enough time/spatial resolution and complexity for restricting the injection of errors to specific bits of a specific frame/s, i.e. to specific bits of the data field and the corresponding CRC of the 2nd frame in this experiment.

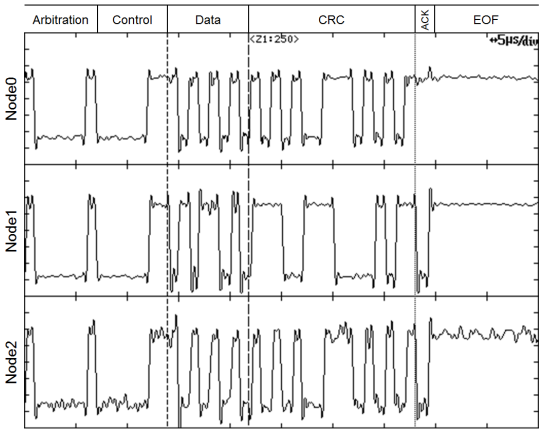

Fig. 5. Integrity error oscilloscope capture.

	Node0	Node1	Node2
1	Tx 010#AA	Rx 010#AA	Rx 010#AA
2	Tx 010#AA	Rx 010#5A	Rx 010#AA
3	Tx 010#AA	Rx 010#AA	Rx 010#AA

TABLE IV
LOG OF THE INTEGRITY EXPERIMENT.

C. Inconsistent scenarios campaign

This Section shows another example of the exclusive capacity of sfiCAN to induce a byzantine fault at an arbitrary application by injecting errors in the channel. However, this section goes much further in order to reveal other important characteristics of sfiCAN.

First, this section demonstrates that the NCC-based architecture of sfiCAN allows building a fully automated testing infrastructure that can autonomously operate not only to inject a given fault scenario, but a whole set of them, i.e. to autonomously perform a *fault-injection campaign*.

Second, it shows the good scalability of sfiCAN. For that purpose, each one of the experiments that compose the campaign involves six nodes and a wider set of messages. In this way we demonstrate that it is possible to easily scale several aspects of the architecture of sfiCAN. Specifically, we included additional software NCC loggers to retrieve data from the new nodes while, on the other hand, we increased the capacity of the centralized fault injector and logger to simultaneously and independently inject into and monitor a higher number of ports. Moreover, we also demonstrate the suitability of the hub and the NCC protocol to communicate the FIMS with a higher number of NCCs.

Finally, this section proves the maturity of sfiCAN for assessing the fault-tolerance mechanisms of a CAN-based system. In particular, the campaign carried out here is devoted to finding out if a combination of up to two erroneous bits happening at the End Of Frame (EOF) can violate the traditional believe that CAN provides *atomic broadcast*, i.e. that in CAN a frame is consistently received by all nodes or by none of them. The results of the campaign demonstrate that sfiCAN easily and quickly detects the two combinations of errors already reported in the literature as causing inconsistency scenarios that violate the atomic broadcast property in CAN, [38] and [39]. In this sense, note that in [40] we carried out a simple single experiment that shows that sfiCAN can be also programmed to directly force the scenario reported in [39].

Figs. 6 and 7 show these two scenarios, which are commonly referred to as *Inconsistent Message Duplicate* (IMD) [38] and *Inconsistent Message Omission* (IMO) [39]. As shown there, each one of these scenarios involves three groups of nodes, namely the transmitter and two groups of receivers called X and Y. An IMD occurs when receivers X detect a dominant bit in the last-but-one bit of the EOF. When so, these receivers reject the frame and transmit an error flag composed of 6 consecutive dominant bits. The first bit of this flag forces the transmitter to reject the frame, to signal its own error flag and, then, to schedule the retransmission of the frame. Conversely, the first bit of the error flag sent by receivers X does not compel receivers Y to reject the frame, but to simply consider that an overload flag is being broadcast (this behavior is known as the *last bit rule* of CAN) [1]. As a consequence, receivers Y will receive the frame twice (the original one and the copy that is retransmitted afterwards).

The IMO scenario is similar to the IMD, the difference is that the transmitter locally encounters an extra error at the last bit of the EOF. This prevents it from detecting the first bit of the error flag sent by receivers X. Thus, it considers the frame as valid, signals an overload flag, an does not retransmits the frame. As a consequence, receivers Y do receive the frame whereas receivers X do not.

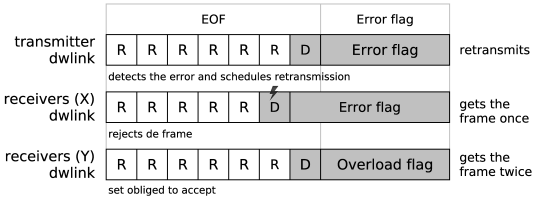

Fig. 6. Inconsistent message duplicate scenario.

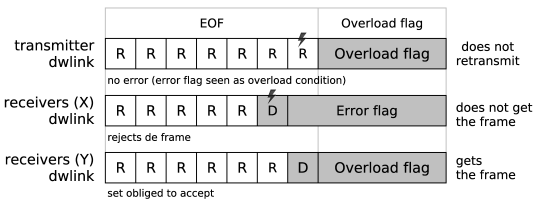

Fig. 7. Inconsistent message omission scenario.

The fault-injection campaign we designed to test if these scenarios can occur in CAN involves one transmitter and five receivers. In order to find out the combinations of errors that lead to an IMD and an IMO, each experiment of the campaign injects a different combination of two erroneous bits during the EOF of a given frame. One of the errors of the combination consists in a dominant bit injected in the downlinks of three of the receivers (which thus play the role of receivers X), whereas the other error is a recessive bit injected in the downlink of the transmitter. No error is injected in the downlinks of the two remaining receivers (they represent the receivers Y).

The remote testing capabilities of sfiCAN allows the FIMS to autonomously conduct the campaign. For this purpose, we programmed the FIMS behavior by means of a simple script. Specifically, the content of this script automatizes the main tasks related to the pre- and post-execution of each fault-injection experiment, that is, the configuration, initialization, finalization and log retrieval. First, for each one of the experiments, the script builds up the fault-injection specification, i.e. what, where and when to inject, and for how long. After that, it configures the Centralized Fault Injector with the corresponding specification and, then, sends the command that triggers the execution of the experiment (See Section III-D). This enables the operation of the Centralized Fault Injector and the Hub's and Nodes' Loggers. Furthermore, it releases nodes from their idle state, so that they start the execution of their applications. Later, when the script considers that the experiment has to finish, it sends an end-of-experiment command that forces all NCCs and applications to stop and wait for further instructions. Finally, at this point, the script retrieves the log information of each NCC Logger and, then, proceeds with the following experiment.

Note that the script must build up the fault-injection specification of all the experiments that are needed to cover all the possible combinations of two errors during the EOF. In order to ease the way in which the script is programmed to do so, we designed two specification templates, i.e. one to configure the injection at the downlinks of receivers X, and another one for configuring the injection at the transmitter downlink. List. 4 shows both of them, namely `fault injection 1` and `fault injection 2` respectively. In each one of these templates the fault-injection parameters are divided into two groups: the set of parameters whose values are common to all experiments, and the set of parameters whose values are specific to each experiment. In this sense, the values of the first type of parameters are predefined and remain constant, whereas the others (surrounded by "<" and ">") are kept undefined so that the script can set them properly.

As shown in List. 4, template `fault injection 1` specifies the dominant bit (line 2) that has to be injected in the downlinks of receivers X, which are connected to ports 3, 4 and 5 (line 3). Since the injection consists in a single bit, the mode is set to single-shot (line 4). As explained in Section IV, this fault-injection mode involves three types of conditions, i.e. `aim`, `fire` and `cease`. The `aim` conditions (lines 6 to 9) are configured to enable the injection as soon as the second frame is being broadcast. Then, the two first `fire` conditions (lines 11 to 12) are set to arm the trigger at the first bit of the EOF

```
1  [fault injection 1]          [fault injection 2]
2  value_type  = dominant       value_type  = recessive
3  target_link = port<3–5>dw    target_link = port0dw
4  mode        = single-shot    mode        = single-shot
5
6  aim_filter  = 0              aim_filter  = 0
7  aim_field   = idle           aim_field   = idle
8  aim_link    = coupled        aim_link    = coupled
9  aim_count   = 2              aim_count   = 2
10
11 fire_field  = eof            fire_field  = eof
12 fire_bit    = 0              fire_bit    = 0
13 fire_offset = <0–6>          fire_offset = <0–6>
14
15 cease_bc    = 1              cease_bc    = 1
```

Listing 4. Campaign templates. Values surrounded by "<" and ">" are set dynamically in every experiment.

field of that frame. The next `fire` condition, i.e. `fire_offset` (line 13), is left unspecified so that the script can set it up to actually start injecting at a specific bit of the EOF. Note that `fire_offset` can take any value between 0 and 6. This makes it possible for the script to cover all the bits of the EOF, that is, from the first to the 7th. Finally, the `cease` condition (line 15) is used to set the size of the injection, measured in number of bits (in this case just one).

Template `fault-injection 2` is designed similarly. It specifies that a recessive bit (line 2) must be injected in the downlink of the transmitter, which is connected to port 0 (line 3). The rest of parameters are configured with the same values as in the other template.

Regarding the operation of the nodes and the hub during each experiment, they behave as follows. The transmitter node constantly sends a frame, in whose data field it includes the number of times it has encountered a successful transmission. This allows to easily identify retransmitted frames within the log. Each receiver simply reads every frame its CAN controller correctly receives from the network. In addition, each node, either a transmitter or a receiver, is provided with an NCC logger that respectively tracks each frame its node correctly transmits or receives, as well as the values of the error counters of the node's CAN controller. As concerns the hub, its injector injects errors as specified, while its logger tracks the content and source of the frames being broadcast. If an error affects a given frame, it also logs the specific bit in which it observed the error. Finally, note that the script acts as an additional node that counts the number of frames being broadcast. When it observes a predefined number of frames, it forces the end of the experiment to proceed with the next one.

As can be inferred from the above discussion concerning the specifications of the fault injections, the script builds up a campaign composed of forty-nine experiments. The execution of the whole campaign at 1Mbs takes about 1.2seg. This time includes the overhead produced by the messages that the FIMS exchanges with the NCCs to configure and coordinate them. This figure shows the good performance of sfiCAN.

The results of the campaign corroborate that an IMD and an IMO happen in two different situations. As expected, an IMD occurs in the experiments in which the hub injects the dominant bit in the 6th bit of the EOF observed by receivers X and, at the same time, it injects the recessive bit in any of the six first bits of the EOF received by the transmitter. Note that all these experiments generate the same scenario reported in the literature as being the cause of an IMD, i.e. receivers X encounter a dominant bit in the 6th bit of the EOF, whereas the transmitter and receivers Y correctly monitor the recessives that compose the EOF up to that bit.

In order to further corroborate the correct identification of the IMD scenario, we reproduced one of these experiments and captured (by means of an oscilloscope) the signals the transmitter and the receivers observe at their downlinks (see Fig. 8). The downlink signals corresponding to the transmitter, receivers X and receivers Y are respectively labelled as `Tx`, `Rx(X)` and `Rx(Y)` at the left of Fig. 8. The top of the figure also specifies the fields of the frame as seen by the transmitter. The most noteworthy aspect of this capture is that it shows how the injected dominant bit compels receivers X to detect an error and, then, to start signalling it at the last bit of the EOF by means of an error flag. Moreover, we also summarize in Table V the data logged during that experiment, which further confirms that the injected error leads to an IMD. Note that, analogously to the capture, the error flag sent by receivers X leads the hub (and thus the transmitter) to detect an error in the 7th bit of the EOF of the second frame (bits are numbered from 0). The log also shows that receivers Y do not observe that error in the second frame and, then, they receive this frame twice when the transmitter retransmits it.

As concerns the IMO, the results corroborate that it happens just in the experiment in which the dominant bit is injected in the downlink of receivers X at the 6th bit of the EOF, while the recessive bit is injected in the transmitter downlink at the 7th of the EOF. Fig. 9 shows an oscilloscope capture of the error scenario provoked by this experiment. As can be seen, the injection of the recessive bit masks the error flag in the transmitter downlink during the 7th bit of the EOF, so that the transmitter is expected to accept the frame and not to retransmit it. The logs' summary in Table V corroborates this behaviour, which leads to an inconsistent state in which only the receivers X miss the reception of the second frame.

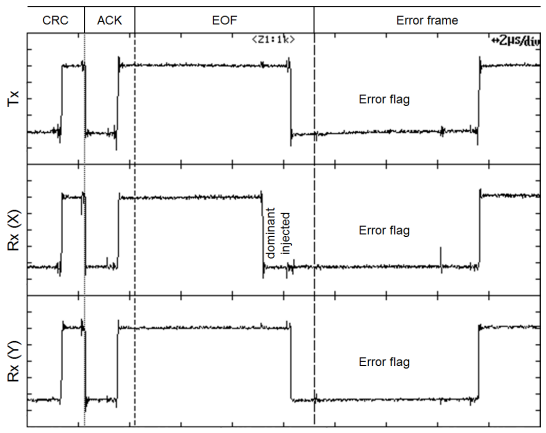

Fig. 8. Inconsistent message duplicate scenario oscilloscope capture.

	Hub	Tx	Rx(X)	Rx(Y)
1	Ok 111#00	Tx 111#00	Rx 111#00	Rx 111#00
2	Error eof(6)	Error	Error	Rx 111#01
3	Ok 111#01	Tx 111#01	Rx 111#01	Rx 111#01
4	Ok 111#02	Tx 111#02	Rx 111#02	Rx 111#02
5	Ok 111#03	Tx 111#03	Rx 111#03	Rx 111#03

TABLE V
LOG OF THE INCONSISTENT MESSAGE DUPLICATE EXPERIMENT.

	Hub	Tx	Rx(X)	Rx(Y)
1	Ok 111#00	Tx 111#00	Rx 111#00	Rx 111#00
2	Error eof(6)	Tx 111#01	Error	Rx 111#01
3	Ok 111#02	Tx 111#02	Rx 111#02	Rx 111#02
4	Ok 111#03	Tx 111#03	Rx 111#03	Rx 111#03
5	Ok 111#04	Tx 111#04	Rx 111#04	Rx 111#04

TABLE VI
LOG OF THE INCONSISTENT MESSAGE OMISSION EXPERIMENT.

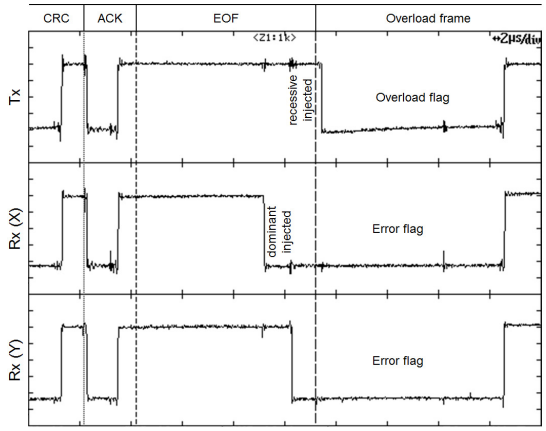

Fig. 9. Inconsistent message omission scenario oscilloscope capture.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents the design and implementation of sfiCAN: the first physical fault-injection infrastructure for CAN that takes full advantage of a star topology. Such topology allows it to overcome the limited space and time resolution of other physical fault injectors for CAN and, thus, to inject fault scenarios that are more complex by targeting any one of the bits each node transmits or receives locally, e.g., sfiCAN can inject faults that are only detected by a subset of the nodes. Moreover, sfiCAN is the only fault injector for CAN that does allow to test the behavior of software under faults without putting any restrictions on the tested software, such as requiring it to be modified or to generate deterministic traffic on the channel. All these capabilities are beyond those of any other injector previously proposed for CAN.

The central element of sfiCAN is a hub that is transparent from the nodes point of view and which is based on the one of [41]. The coupling schema of this hub is what makes it possible for sfiCAN to achieve its high spatial/time resolution. The only limitation is that it requires to include an extra COTS transceiver per node, which still may pose some doubts about the practicality of sfiCAN in industries where costs related to compatibility with hardware configuration of legacy nodes is an issue.

The rest of the fault-injection infrastructure is distributed as a set of components called NCCs whose operation can be configured and coordinated remotely, using native CAN, from a management station connected to that hub. Each fault injection experiment is configured using many parameters, such as different trigger and end conditions, which makes the specification of fault scenarios highly flexible and potent.

The main NCCs are a fault injector and a logger, synthesized together with the hub in the same FPGA. Both have access to every bit each node transmits/receives through its corresponding uplink/downlink. This allows them to inject erroneous bits and observe the subsequent reaction of every node with high spatial and time resolution. Additionally, each node's application has an embedded software NCC logger, which retrieves information about the application's actions and the status of its CAN controller. This NCC-based architecture allows to build a fully automated testing infrastructure for CAN-based systems and protocols that is scalable. More fault-injection and monitoring features can be added inside the nodes or other devices, e.g, the injector within the hub can be extended to inject application-specific faults such as babbling-idiot faults, and the nodes' logger can be programmed to monitor more information of the application itself.

Finally, note that the general idea of using a star-based centralized fault injector, together with NCCs, may also be used to evaluate the dependability of distributed applications that use an underlying communication system other than CAN.

ACKNOWLEDGEMENTS

This work was supported by the Spanish *Ministerio de Ciencia e Innovación* with grant DPI2008-02195, by the Spanish *Ministerio de Economía y Competividad* with grants DPI2011-22992 and BES-2012-052040, by FEDER funding, and by *Govern Balear* (Ref 71/2011).

REFERENCES

[1] *ISO11898-1. Controller Area Network (CAN) - Part 1: Data link layer and physical signalling.*, ISO Std., 2003.
[2] H. Zeltwanger, "Controller Area Network — introduced 25 years ago," *CAN Newsletter*, pp. 18–20, March 2011.
[3] S. Kim, E. Lee, M. Choi, H. Jeong, and S. Seo, "Design Optimization of Vehicle Control Networks," *IEEE Transactions on Vehicular Technology*, vol. 60, no. 7, pp. 3002–3016, 2011.
[4] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proceedings of the IEEE*, vol. 93, no. 6, 2005.
[5] P. Lanigan, P. Narasimhan, and T. Fuhrman, "Experiences with a CANoe-based fault injection framework for AUTOSAR," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2010, pp. 569–574.

[6] R. A. Gupta and M.-Y. Chow, "Networked Control System: Overview and Research Trends," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 7, pp. 2527–2535, Jul. 2010.

[7] F. Baronti, E. Petri, S. Saponara, L. Fanucci, R. Roncella, R. Saletti, P. Abramo, and R. Serventi, "Design and Verification of Hardware Building Blocks for High-Speed and Fault-Tolerant In-Vehicle Networks," *IEEE Trans. on Industrial Electronics*, vol. 58, no. 3, pp. 792–801, 2011.

[8] H. Kimm and H.-s. Ham, "Integrated Fault Tolerant System for Automotive Bus Networks," in *2010 2th International Conf. on Computer Engineering and Applications*. Ieee, 2010, pp. 486–490.

[9] United States Environmental Protection Agency, "Control of air pollution from new motor vehicles and new motor vehicle engines; modification of federal on-board diagnostic regulations for: light-duty vehicles, light-duty trucks, medium duty passenger vehicles, complete heavy-duty vehicles and engines intended for use in heavy duty vehicles weighing 14,000 pounds GVWR or less," *United States Federal Register*, vol. 70, no. 243, pp. 75 403–75 411, 2005.

[10] The Commission of the European Communities, "Commission Directive 2002/80/EC of 3 October 2002 - adapting to technical progress Council Directive 70/220/EEC relating to measures to be taken against air pollution by emissions from motor vehicles," *Official Journal of the European Communities*, vol. 291, pp. 20–56, 2002.

[11] J. Munoz-Castaner, R. Asorey-Cacheda, F. Gil-Castineira, F. Gonzalez-Castano, and P. Rodriguez-Hernandez, "A Review of Aeronautical Electronics and Its Parallelism With Automotive Electronics," *IEEE Trans. on Industrial Electronics*, vol. 58, no. 7, pp. 3090–3100, 2011.

[12] T. Nolte, M. Nolin, and H. Hansson, "Real-Time Server-Based Communication With CAN," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 3, pp. 192–201, Aug. 2005.

[13] B. Gaujal and N. Navet, "Fault confinement mechanisms on CAN: analysis and improvements," *IEEE Transactions on Vehicular Technology*, vol. 54, no. 3, pp. 1103–1113, 2005.

[14] J. Rufino, C. Almeida, P. Verissimo, and G. Arroz, "Enforcing Dependability and Timeliness in Controller Area Networks," in *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*. IEEE, Nov. 2006, pp. 3755–3760.

[15] J. Ferreira, L. Almeida, J. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza, "Combining Operational Flexibility and Dependability in FTT-CAN," *IEEE Transactions on Industrial Informatics*, vol. 2, no. 2, pp. 95–102, May 2006.

[16] K. Schmidt and E. G. Schmidt, "Systematic Message Schedule Construction for Time-Triggered CAN," *IEEE Transactions on Vehicular Technology*, vol. 56, no. 6, pp. 3431–3441, 2007.

[17] G. Buja, J. R. Pimentel, and A. Zuccollo, "Overcoming Babbling-Idiot Failures in CAN Networks: A Simple and Effective Bus Guardian Solution for the FlexCAN Architecture," *IEEE Transactions on Industrial Informatics*, vol. 3, no. 3, pp. 225–233, Aug. 2007.

[18] M. Short and M. J. Pont, "Fault-Tolerant Time-Triggered Communication Using CAN," *IEEE Transactions on Industrial Informatics*, vol. 3, no. 2, pp. 131–142, May 2007.

[19] B. Hall, M. Paulitsch, K. Driscoll, and H. Sivencrona, "ESCAPE CAN limitations," *SAE Trans. J. Passenger Cars - Electron. Elect. Syst*, vol. 116, pp. 422–429, 2008.

[20] J. Pimentel, J. Proenza, L. Almeida, G. Rodriguez-Navas, M. Barranco, and J. Ferreira, "Dependable Automotive CANs," in *Automotive Embedded Systems Handbook*, N. Navet and F. Simonot-Lion, Eds. CRC Press, 2008, ch. 6, pp. 1–56.

[21] G. Rodriguez-Navas, "Orthogonal, Fault-Tolerant, and High-Precision Clock Synchronization for the Controller Area Network," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 2, pp. 92–101, 2008.

[22] H. Zeng, M. D. Natale, P. Giusto, and A. Sangiovanni-vincentelli, "Stochastic Analysis of CAN-Based Real-Time Automotive Systems," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 4, pp. 388–401, Nov. 2009.

[23] R. Miucic, S. M. Mahmud, and Z. Popovic, "An Enhanced Data-Reduction Algorithm for Event-Triggered Networks," *IEEE Transactions on Vehicular Technology*, vol. 58, no. 6, pp. 2663–2678, 2009.

[24] T. Herpel, K.-S. Hielscher, U. Klehmet, and R. German, "Stochastic and deterministic performance evaluation of automotive CAN communication," *Computer Networks*, vol. 53, no. 8, pp. 1171–1185, Jun. 2009.

[25] M. Nahas, M. J. Pont, and M. Short, "Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol," *Journal of Systems Architecture*, vol. 55, no. 5-6, pp. 344–354, May 2009.

[26] T. Hoppe, S. Kiltz, and J. Dittmann, "Security threats to automotive CAN networks—Practical examples and selected short-term countermeasures," *Reliability Engineering & System Safety*, vol. 96, no. 1, pp. 11–25, Jan. 2011.

[27] M. Barranco, J. Proenza, and L. Almeida, "Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate," *Computer*, vol. 42, pp. 66–73, May 2009.

[28] ——, "Quantitative Comparison of the Error-Containment Capabilities of a Bus and a Star Topology in CAN Networks," *IEEE Transactions on Industrial Electronics*, vol. 53, no. 3, pp. 802–803, 2011.

[29] J. Suwatthikul, R. McMurran, and R. Jones, "In-vehicle network level fault diagnostics using fuzzy inference systems," *Applied Soft Computing*, vol. 11, no. 4, pp. 3709–3719, Jun. 2011.

[30] A. Monot, N. Navet, and B. Bavoux, "Impact of clock drifts on CAN frame response time distributions," in *16th IEEE International Conference on Emerging Technologies and Factory Automation*, 2011.

[31] H. Mei-Chen, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[32] M. Kim, S. Lee, and K. Lee, "Experimental Performance Evaluation of Smoothing Predictive Redundancy Using Embedded Microcontroller Unit," *IEEE Trans. on Industrial Electronics*, vol. 58, no. 3, pp. 784–791, 2011.

[33] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin, "Evaluation of fault handling of the time-triggered architecture with bus and star topology," in *Proceedings. 2003 International Conference on Dependable Systems and Networks*, 2003, pp. 123–132.

[34] E. Armengaud, A. Steininger, and M. Horauer, "Towards a Systematic Test for Embedded Automotive Communication Systems," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 3, pp. 146–155, Aug. 2008.

[35] I. Park and M. Sunwoo, "FlexRay Network Parameter Optimization Method for Automotive Applications," *IEEE Trans. on Industrial Electronics*, vol. 58, no. 4, pp. 1449–1459, 2011.

[36] P. Ferrari, A. Flammini, D. Marioli, and A. Taroni, "A Distributed Instrument for Performance Analysis of Real-Time Ethernet Networks," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 1, pp. 16–25, 2008.

[37] International Electrotechnical Commission, "IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems," 1999.

[38] J. Rufino, P. Veríssimo, and A. Guillerme, "Fault-Tolerant Broadcasts in CAN," in *Digest of Papers 28th Annual Int. Symposium on Fault-Tolerant Computing*, 1998, pp. 150–159.

[39] J. Proenza and J. Miro-Julia, "MajorCAN: A Modification to the Controller Area Network Protocol to Achieve Atomic Broadcast," *IEEE International Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.

[40] D. Gessner, M. Barranco, A. Ballesteros, and J. Proenza, "Designing sfiCAN: a star-based physical fault injector for CAN," in *16th IEEE International Conference on Emerging Technologies and Factory Automation*, 2011.

[41] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida, "An Active Star Topology for Improving Fault Confinement in CAN Networks," *IEEE Transactions on Industrial Informatics*, vol. 2, no. 2, pp. 78–85, May 2006.

[42] "FlexRay Communications System Preliminary Central Bus Guardian Specification Version 2.0.9," 2005.

[43] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.

[44] Vector Informatik GmbH, "Product Information CANoe," 2013.

[45] G. Rodríguez-Navas, J. Jiménez, and J. Proenza, "An architecture for physical injection of complex fault scenarios in CAN networks," in *Proc. 9th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, vol. 2, 2003.

[46] Vector Informatik GmbH, "CANstress, User Manual, Version 2.1," 2006.

[47] J. P. Acle, M. S. Reorda, and M. Violante, "Early, accurate dependability analysis of CAN-based networked systems," *IEEE Design & Test of Computers*, vol. 23, no. 1, pp. 38–45, Jan. 2006.

[48] H. Webermann and A. Block, "CAN Error Injection, a Simple but Versatile Approach," in *13rd International CAN Conference (iCC 2012)*, 2012, pp. 14–19.

[49] P. Koopman, E. Tran, and G. Hendrey, "Towards Middleware Fault Injection for Automotive Networks," in *28th Int. Symposium on Fault-Tolerant Computing (FTCS-28)*, 1998.

[50] M. S. Reorda and M. Violante, "On-line analysis and perturbation of CAN networks," in *Proc. 19th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, 2004.

[51] F. Luo, M. Mo, C. Liu, and Z. Huang, "CAN disturbances generator development," in *2009 IEEE Vehicle Power and Propulsion Conference*. IEEE, 2009, pp. 1587–1591.

[52] J. Novak, A. Fried, and M. Vacek, "CAN generator and error injector," in *9th International Conference on Electronics, Circuits and Systems (ICECS)*, vol. 3. IEEE, 2002, pp. 967–970.
[53] S. Khoshbakht and H. Zarandi, "Soft error propagations and effects analysis on CAN controller," in *IEEE International Conference on Automation Quality and Testing Robotics (AQTR)*, vol. 2. IEEE, 2010, pp. 1–7.
[54] H. Zarandi, S. Miremadi, and A. Ejlali, "Dependability analysis using a fault injection tool based on synthesizability of HDL models," in *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, 2003, pp. 485–492.
[55] F. Corno, S. Tosato, and P. Gabrielli, "System-level analysis of fault effects in an automotive environment," in *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE Comput. Soc, 2003, pp. 529–536.
[56] "International Standard ISO/IEC 14977 - Information technology - Syntactic metalanguage - Extended BNF," 1996.
[57] Microchip Technology, "dsPIC30F6011A/6012A/6013A/6014A Data Sheet," 2008.

David Gessner received the first degree in informatics engineering and a master in information and communication technologies from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2010 and 2011, respectively.

He is currently pursuing the Ph.D. degree in computer science at the UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, dependable communication topologies, and field-bus networks such as CAN.

Manuel Barranco received the first degree in informatics engineering and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2003 and 2010, respectively.

He is currently a lecturer in the Department of Mathematics and Informatics at UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, clock synchronization, dependable communication topologies, and field-bus networks such as CAN.

Alberto Ballesteros received the first degree in informatics engineering from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2012.

He is currently hired as a technician in the Department of Mathematics and Informatics at UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems and dependable communications in distributed embedded systems.

Julián Proenza received the first degree in physics and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 1989 and 2007, respectively.

He is currently holding a permanent position as a lecturer in the Department of Mathematics and Informatics at UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, clock synchronization, dependable communication topologies, and field-bus networks such as CAN.

Dr. Proenza is a Senior Member of the IEEE Industrial Electronics Society since 2013.